Kernel Implementation: Page table structures

Cristan Szmajda

cls@cse.unsw.edu.au

Virtual memory

Almost all modern operating systems support virtual memory. VM lets you:

- run applications larger than physical memory
- make best use of physical memory
- protect applications from each other (and themselves!)

But, paging virtual memory has some unavoidable overheads:

- translation lookaside buffer (TLB)
- TLB misses
- page table
- page faults

In the 1980's, these overheads were typically around 5% to 8% (Clark & Emer 1985).

Suddenly in the mid-1990's, studies start to report TLB refill overheads of 30% to 50%, and even 80%.

What went wrong?

- ↑ increasing MHz
- ↑ increasing MBytes
- ↑ increasing instructions per cycle (superscalar, VLIW, etc.)
- ↑ more address bits (64-bit addresses)
- ↑ higher miss penalty (pipeline & exception costs)
  - i80386: 9 to 13 cycles
  - VAX-11/780: 17 to 23 cycles
  - Pentium 4: 31 to 48 cycles (assuming L2 cache hits)
  - PowerPC 604: 120 cycles (assuming level 2 TLB hit)

TLBs aren’t getting bigger and faster as fast as CPU and RAM.

Why not just make bigger and faster TLBs?

- large CAMs are slow and hot
- often flushed (context switch, address space teardown, protection change, etc.)
- MHz, MBytes, and caches sell computers, not TLBs

Why not just increase the page size?

- fragmentation
- I/O latency
- inertia

In many processors, TLB thrashing is a bottleneck.

System impact

Some system features exacerbate TLB load.

- service decomposition
  - user-level daemons
  - micro-kernels
- sparse address space layout
- shared memory
  - increases fragmentation
  - reduces locality
  - requires more TLB entries to cover the available RAM

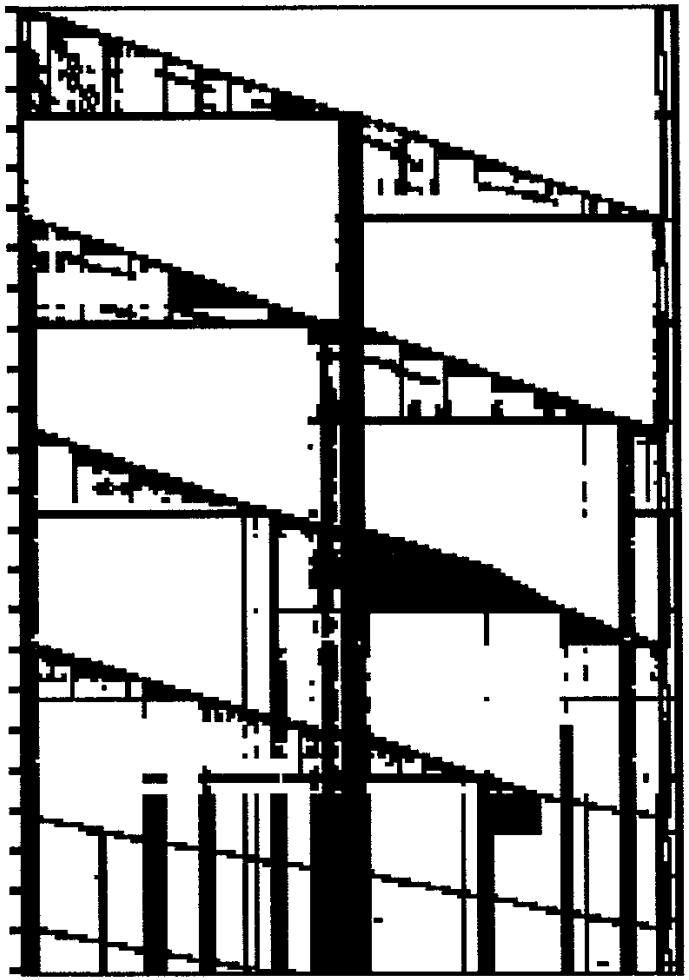
Micro-kernel based systems and SASOSes particularly suffer.

Application impact

Some features of recent languages and applications reduce locality:

- bloat
- dynamic libraries
- indirection
- garbage collection

Chen, Borg, and Jouppi (1992) trace a process undergoing garbage collection.



tree-data
instruction range: 0-2410000000
page range:10000-10a5c
block = 10000000 instructions x 0x10000 bytes

Hardware solutions

- shared TLB tags (StrongARM, PA-RISC, IA-64)
- virtual caches
- super-pages

machine	ITLB	DTLB	page sizes
StrongARM	32	32	4k, 64k, 1M
Pentium III	32	64	4k, 4M
Itanium	64	96	4k, 8k, 16k, 64k, 256k, 1M, 4M, 16M, 64M, 256M, 4G
Alpha 21264	128	128	8k, 64k, 512k, 4M
UltraSPARC	64		8k, 64k, 512k, 4M
MIPS R4000	96		4k, 16k, 64k, 256k, 1M, 4M, 16M
PowerPC 601	256		4k

Software solutions

- optimize page table lookup
- cross-link page tables for shared memory

Page table structures

TLB miss overhead is directly limited by page table performance.

Classical page table structures were designed for

- 32-bit address spaces, and
- the Unix two-segment model.

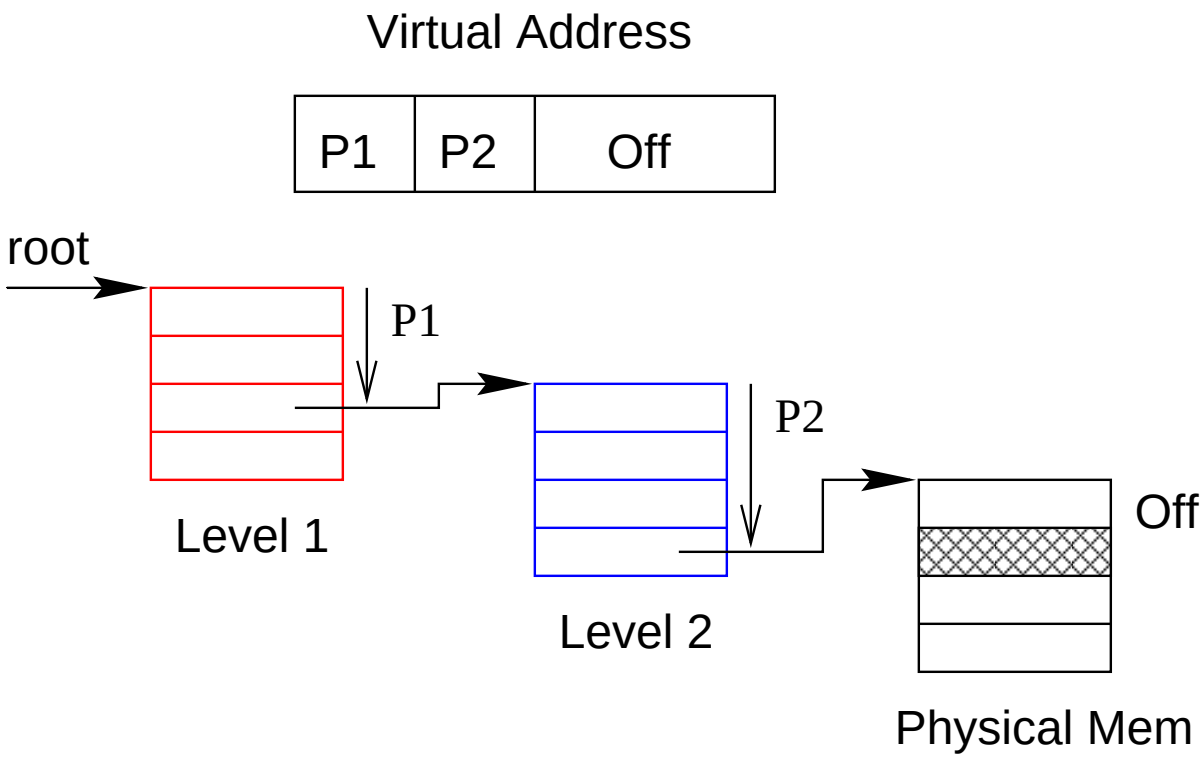
How well do they perform in:

- large (64-bit) address spaces?
- sparse address-space distributions?
- micro-kernel system structures?

Requirements

- time
- space
- establishment, tear-down cost
- cache friendliness
- support for sharing/aliasing
- support for mixed page sizes
- support for operations on large regions

Multi-level page table

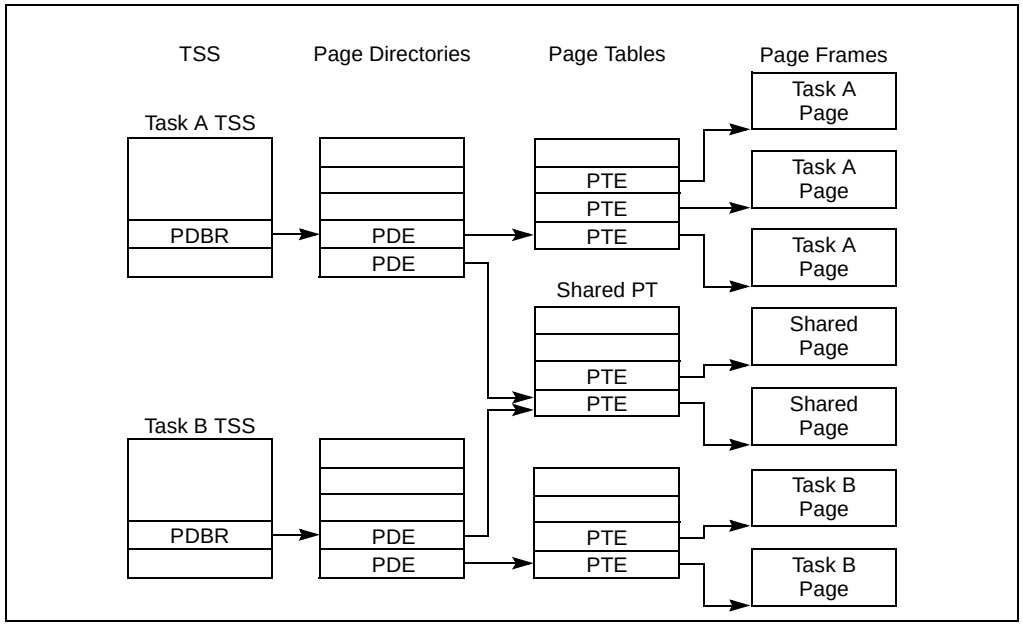


- used on many 32-bit processors (Pentium, StrongARM, etc.)
- require 4–5 levels for 64-bit address space (AMD Hammer)
- used in Linux (on all platforms)

Multi-level page table

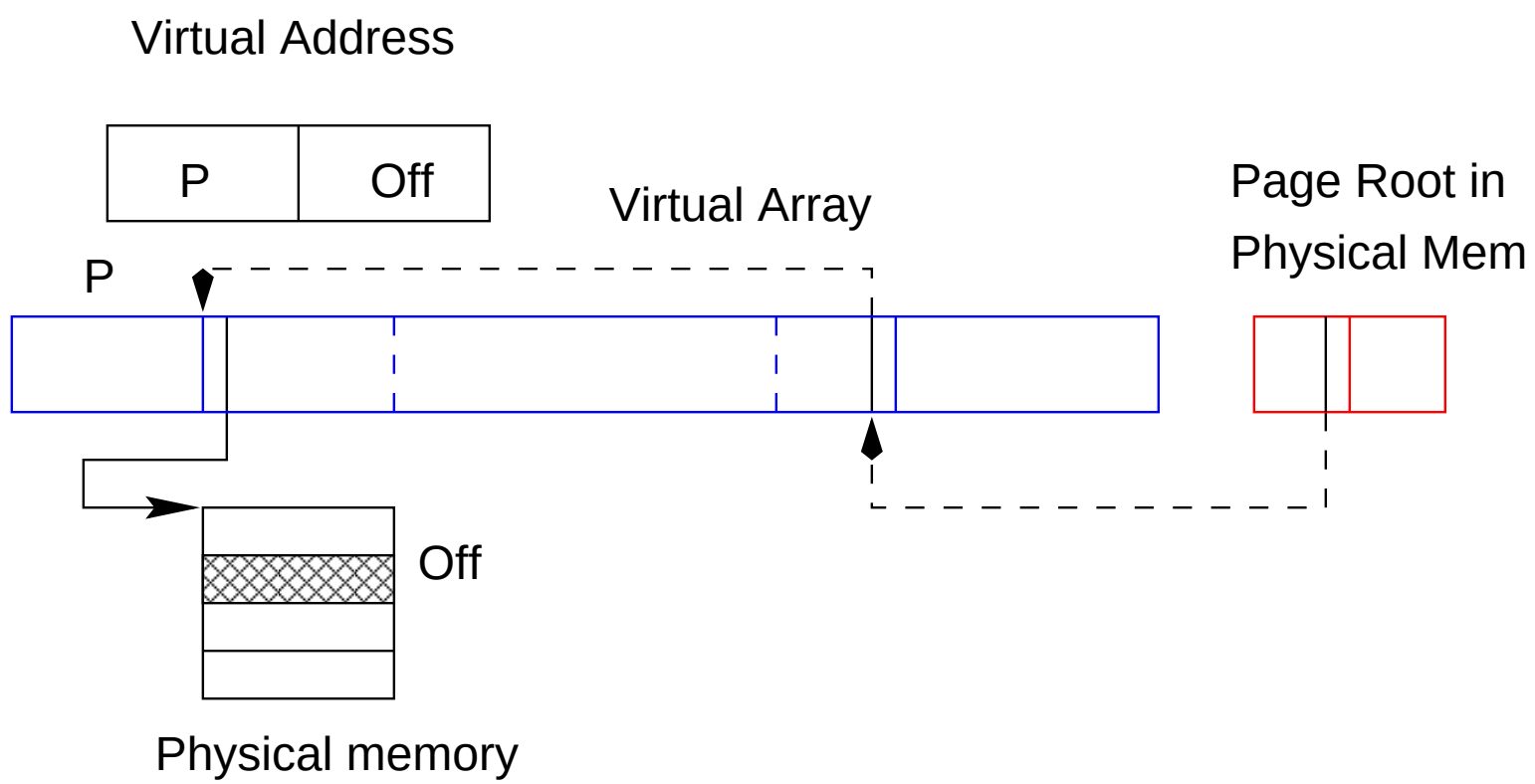
Advantages:

- can support superpages efficiently
- can support page table sharing



- significant alignment restrictions on this sharing and superpage support

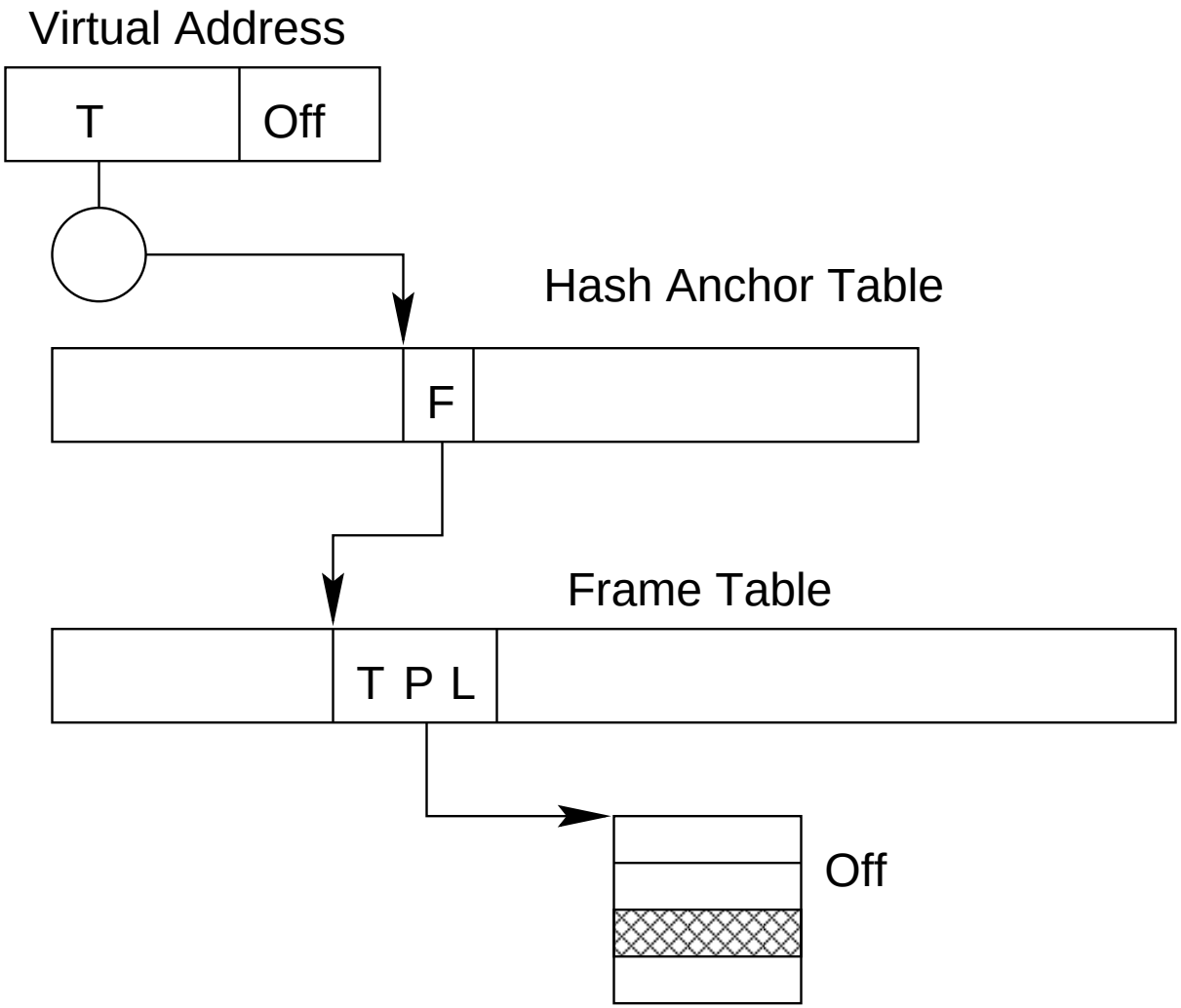
Virtual linear page table



Equivalent to MPT, but:

- better best-case lookup time
- steals TLB entries from application
- requires nested exception handling

Inverted page table



Inverted page table

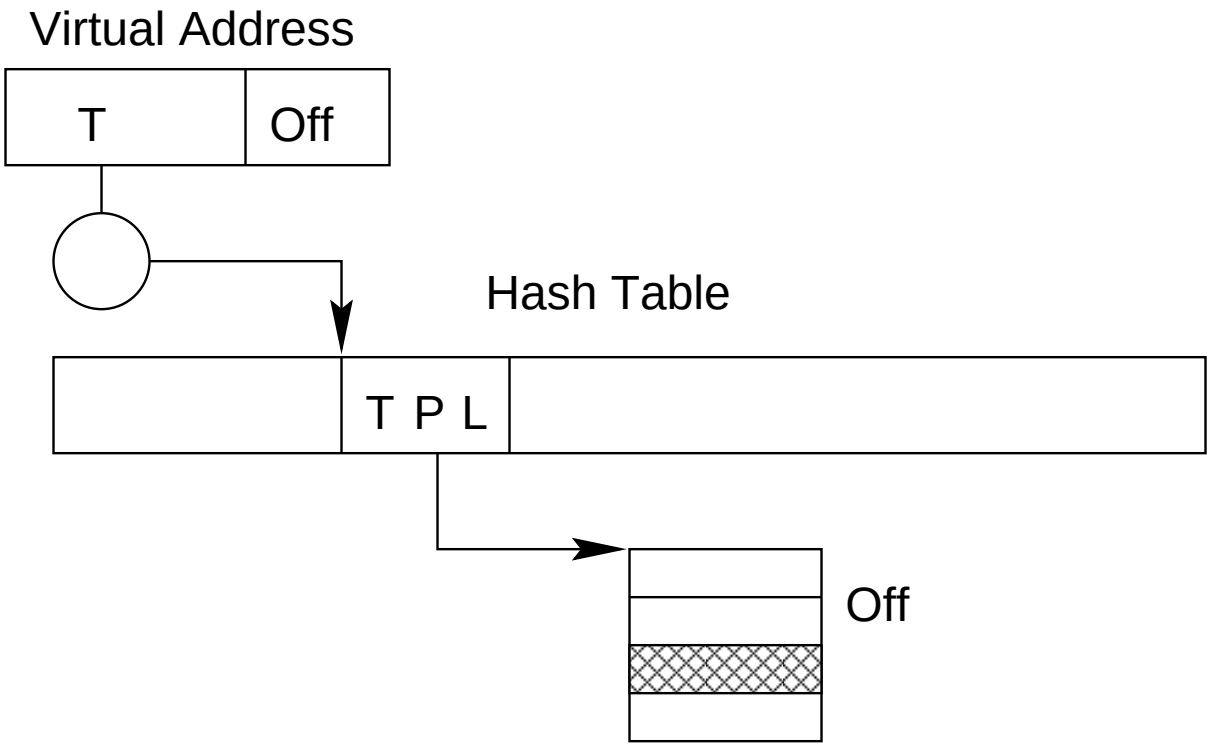
Advantages:

- scales with physical, not virtual, memory size
- no problem with virtual sparsity
- one IPT per system, not per address space
- PTEs bigger as need to store tag
- system needs a frame table anyway

Disadvantages:

- newer machines have sparse physical address space
- difficult to support super-pages
- sharing is hard

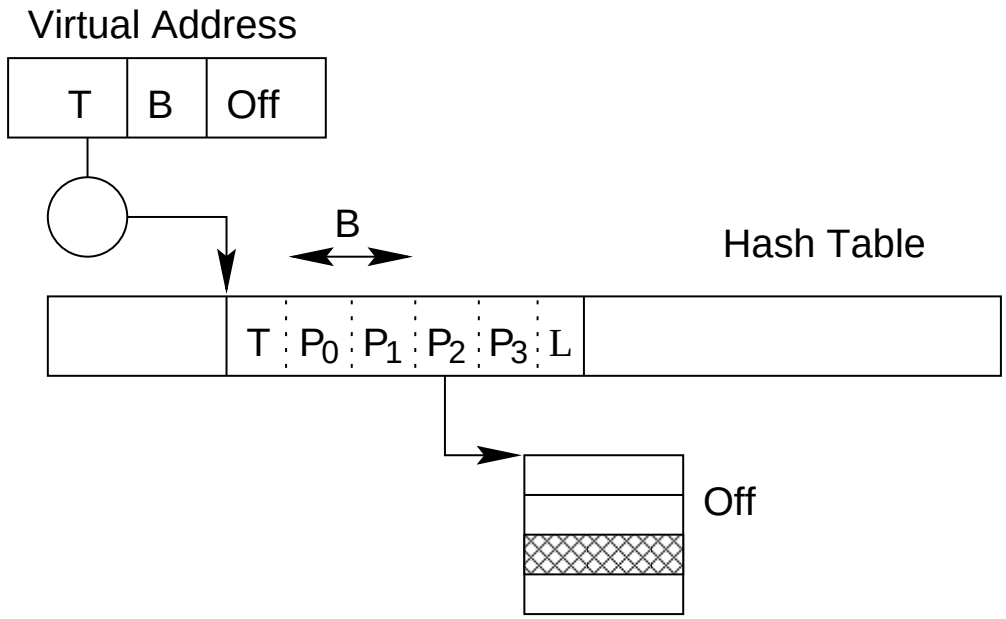
Hashed page table



Very similar to IPT.

Clustering

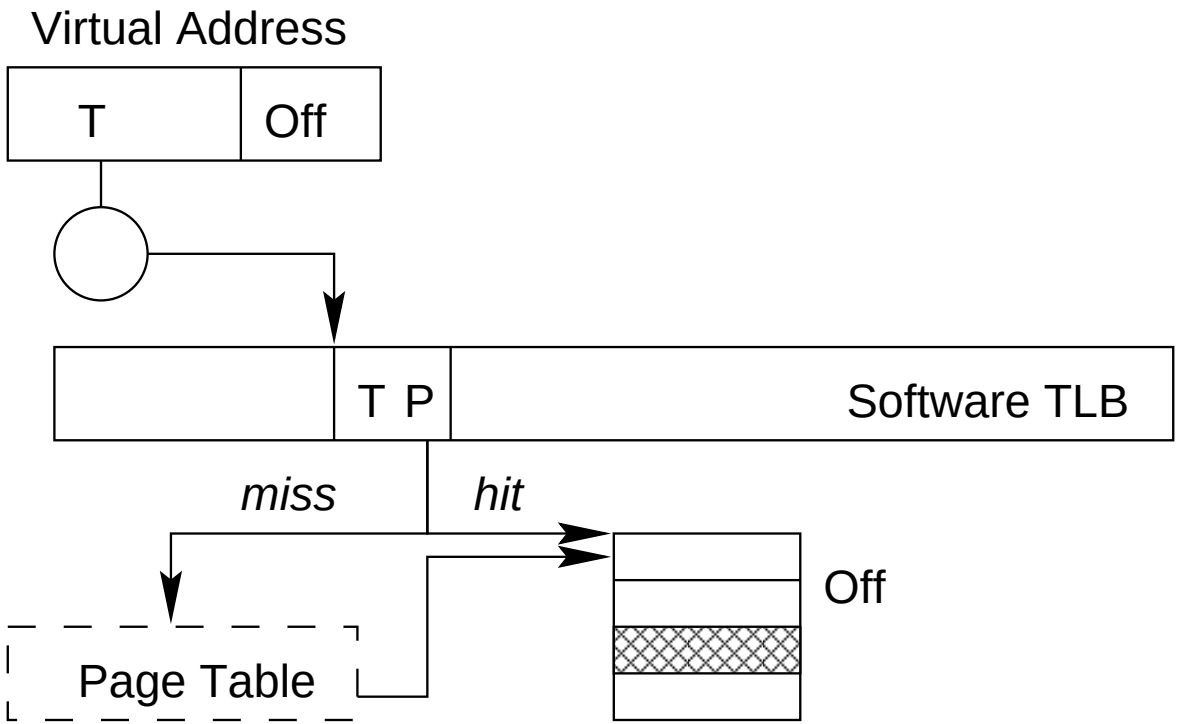
Clustering is a page table optimization which can in principle be applied to any page table structure.



- store multiple pages per PTE
- load multiple pages into TLB per miss
- improves performance in presence of spatial locality
- used in MIPS R4000 hardware TLB entry

Level 2 TLB

- a direct-mapped cache of TLB entries in main memory
- fast lookup; can achieve >95% hit rate
- also called *software TLB* or *TLB cache*

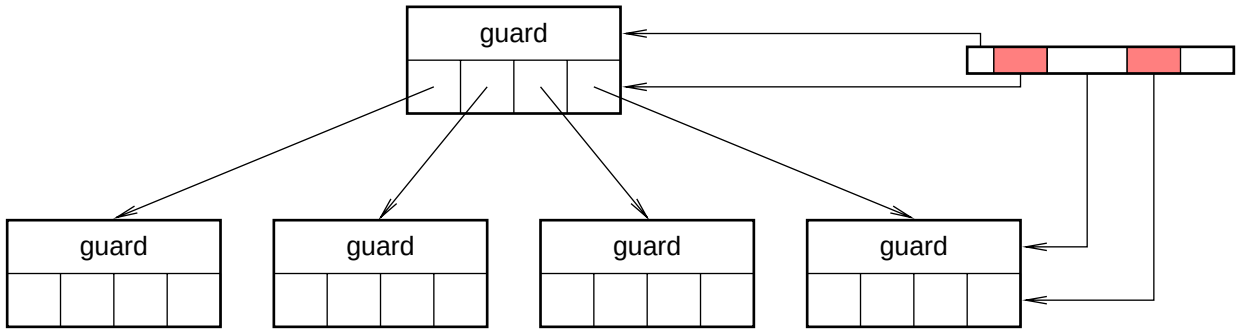


- simple enough for hardware implementation
- difficult to support super-pages

Guarded page table

In large address spaces, MPT often creates page table levels with only one valid entry.

- **idea**: bypass these tables
- some address bits are not used to index any table: check these bits during lookup
- skipped bits called a *guard*
- technique also called *path compression*



Invented by Liedtke (1994), inventor of L4.

TLB performance studies

How well do these page tables perform?

reference	CPU	OS	page table	bench-mark	TLB miss penalty
Clark & Emer (1985)	VAX-11/780	VMS 2	VLA2	*TS1*	6.6%
				TS2	6.4%
				EDU	6.0%
				SCI	5.5%
				COM	6.8%
Nagle et al. (1993)	MIPS R2000	Ultrix 3.1	VLA2	*mixed*	2.03%
		OSF/1 1.0	VLA2½		5.81%
		Mach 3.0	VLA2½		8.21%

reference	CPU	OS	page table	bench-mark	TLB miss penalty
Huck & Hays (1993)	PA-RISC	HP-UX	IPT	*matrix*	45%
				nasker	18%
				OLTP1	12%
				gcc	6%
			HPT	*matrix*	39%
				nasker	15%
				OLTP1	9%
				gcc	4%
Talluri & Hill (1994)	UltraSPARC	Solaris 2.1	HPT	*coral*	50%
				nasa7	40%
				compress	26%
				mp3d	11%
				gcc	3%

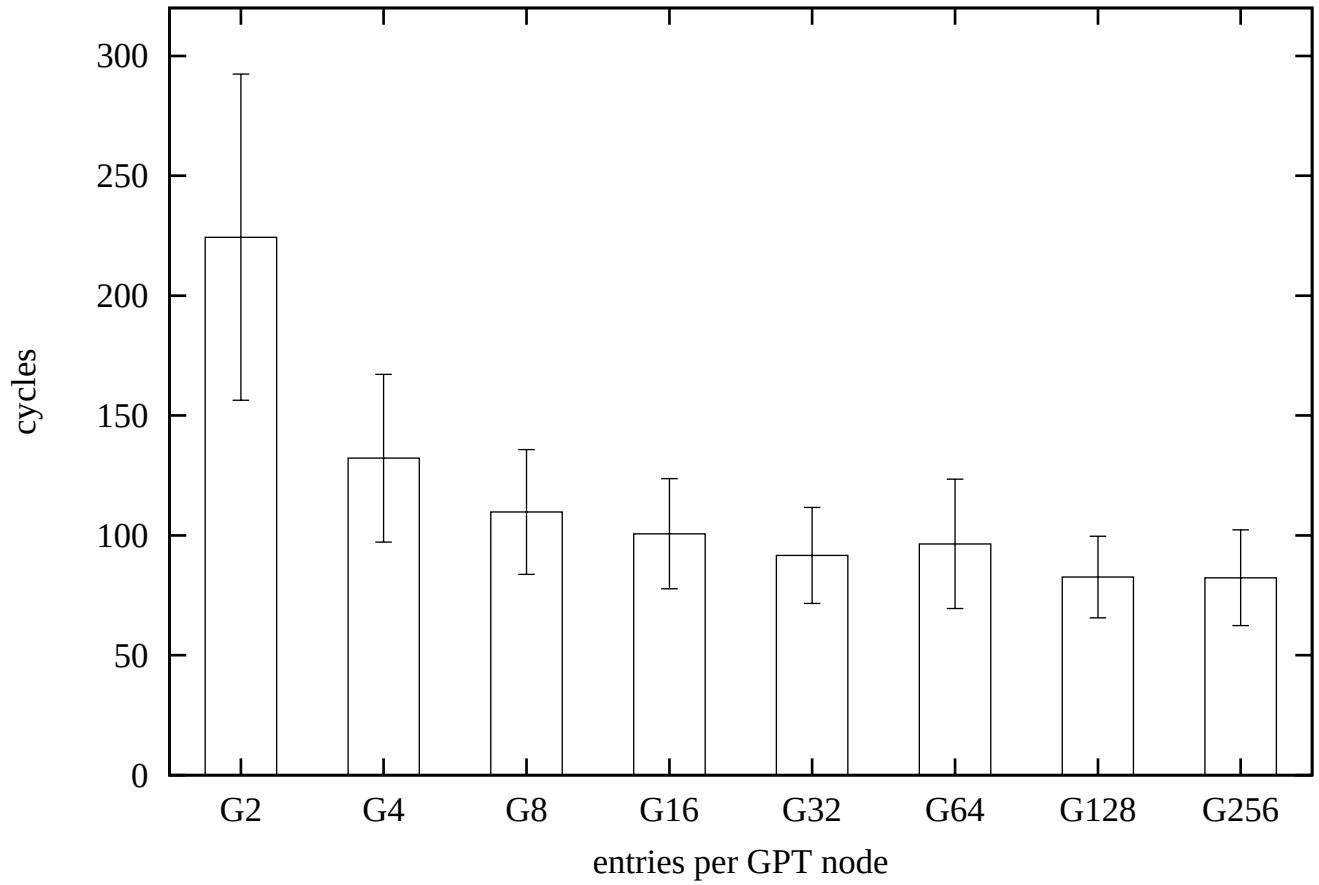
reference	CPU	OS	page table	bench-mark	TLB miss penalty
Romer et al. (1995)	Alpha 21064	OSF/1 2.1	VLA2½	*coral* *compress* *spice* *gcc*	41.4% 35.2% 9.4% 5.2%
Subramanian et al. (1998)	PA-RISC	HP-UX	HPT	*Verilog* *apsi* *compress*	49% 31% 15%
Elphinstone (1999)	MIPS R4700	L4/MIPS	GPT16	*c4* *gcc* *compress* *wave5*	32.4% 17.9% 14.0% 9.3%
			128k STLB	*c4* *gcc* *compress* *wave5*	14.4% 5.9% 4.9% 3.1%

GPT performance

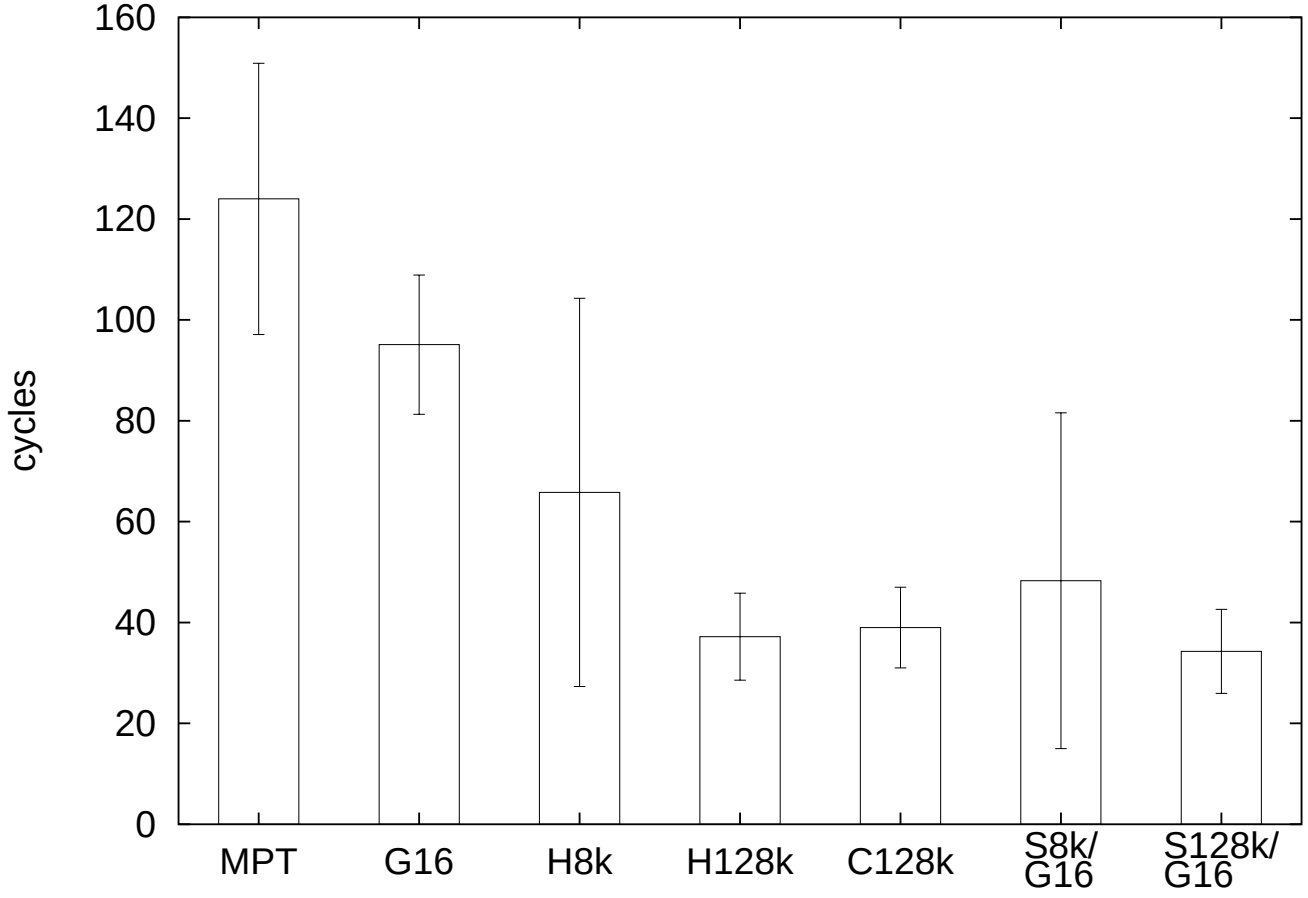
Elphinstone (1999) studied GPT and various other page tables, using L4/MIPS as a testbed.

source	name	size (M)	type	remarks
	go	0.8	I	game of go
	swim	14.2	F	PDE solver
SPEC	gcc	9.3	I	GNU C compiler
CPU95	compress	34.9	I	file (un)compression
	apsi	2.2	F	PDE solver
	wave5	40.4	F	PDE solver
	c4	5.1	I	game of connect four
	nsieve	4.9	I	prime number generator
Alburto	heapsort	4.0	I	sorting large arrays
	mm	7.7	F	matrix multiply
	tfftdp	4.0	F	fast fourier transform

GPT refill time

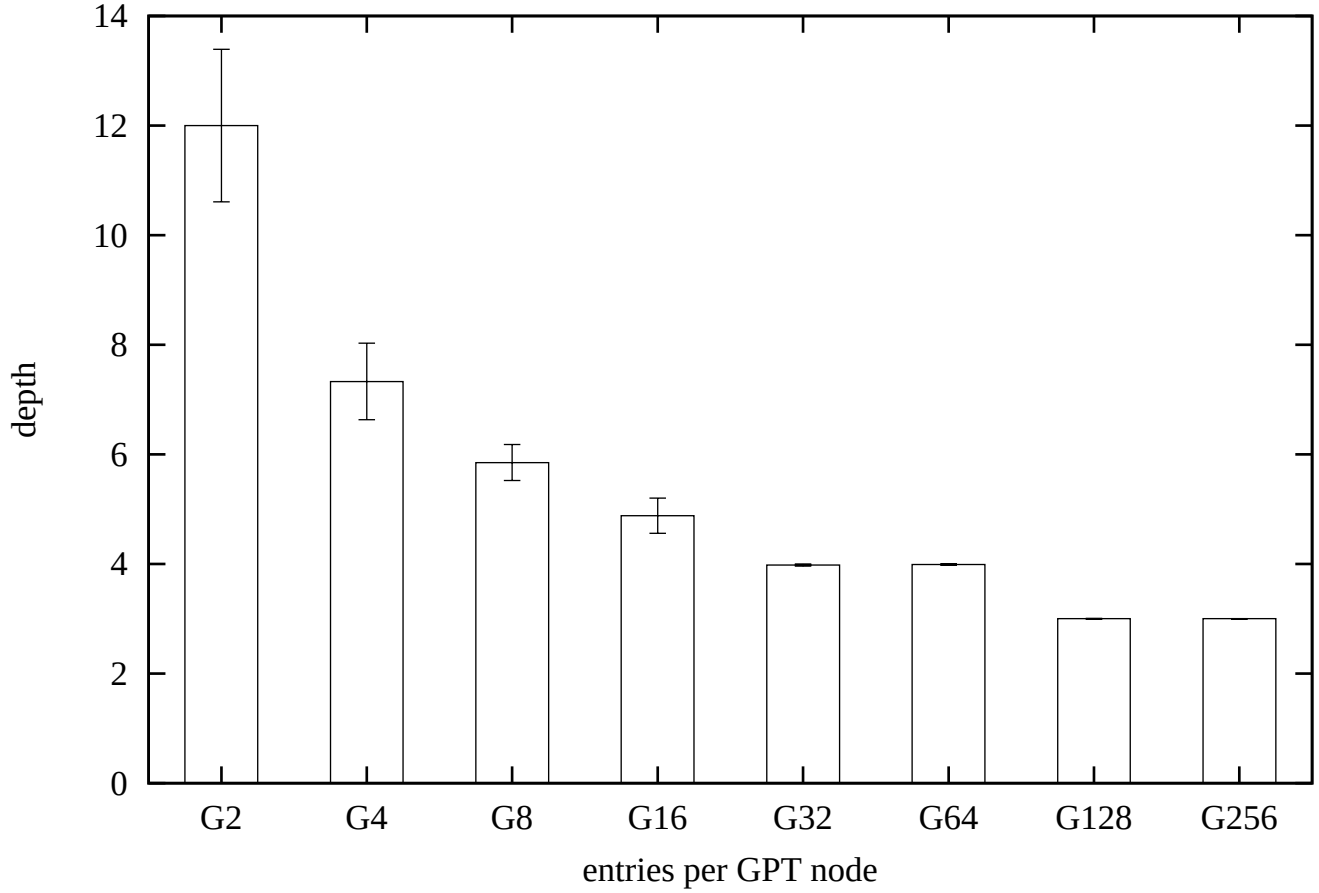


GPT versus other page tables



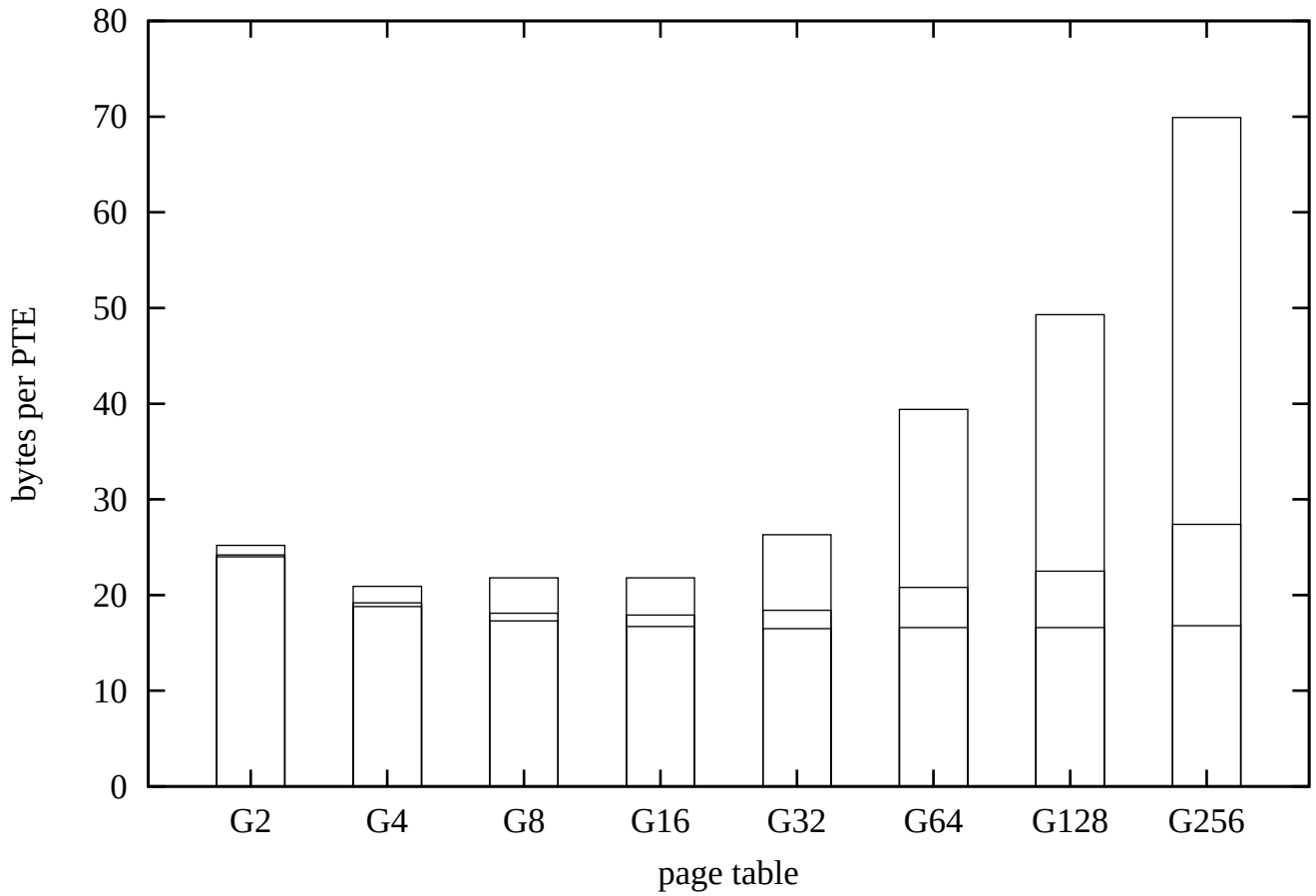
GPT depth

Compare average GPT depth with (fixed) MPT depth.

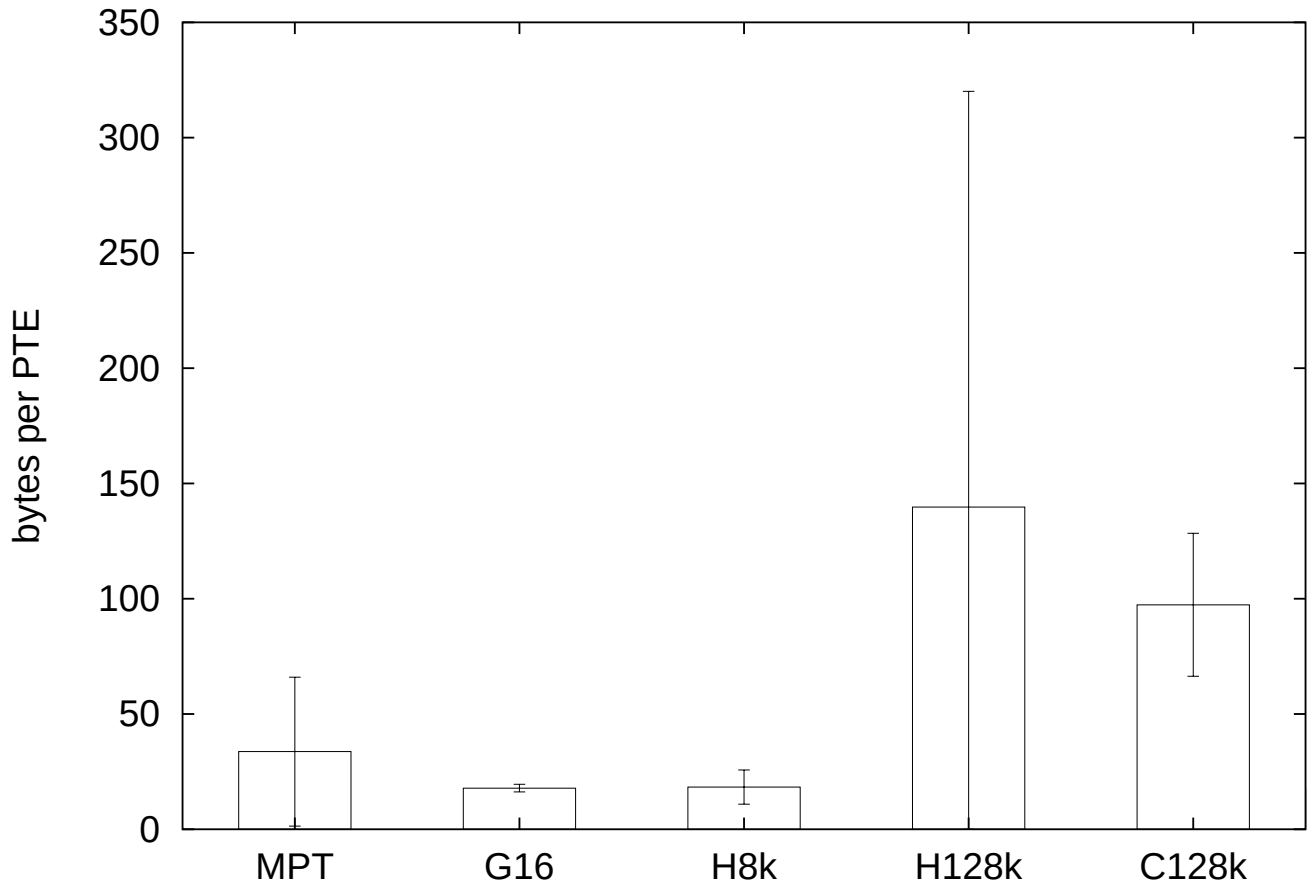


GPT space

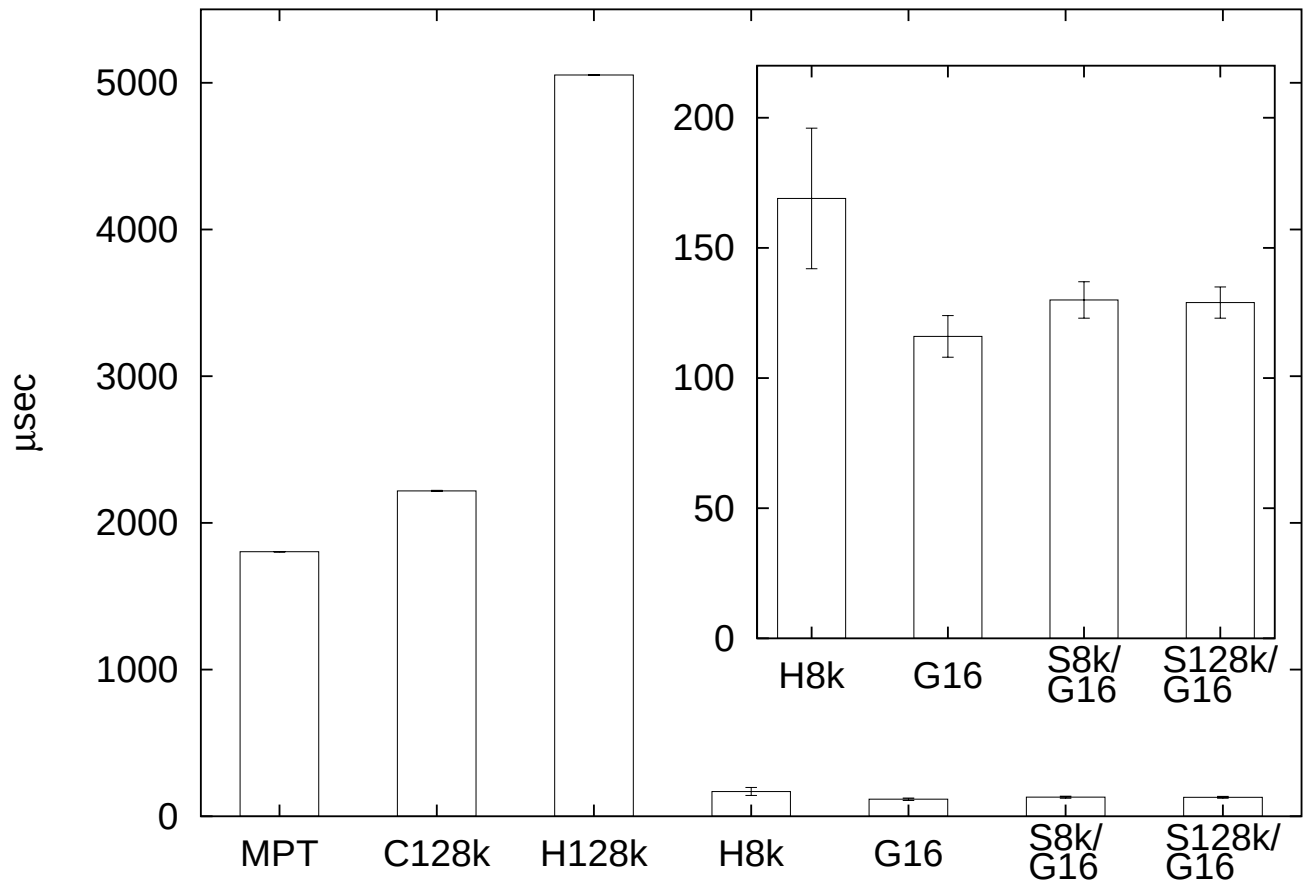
Compare GPT storage requirements with expected MPT storage requirements.



GPT versus other page tables

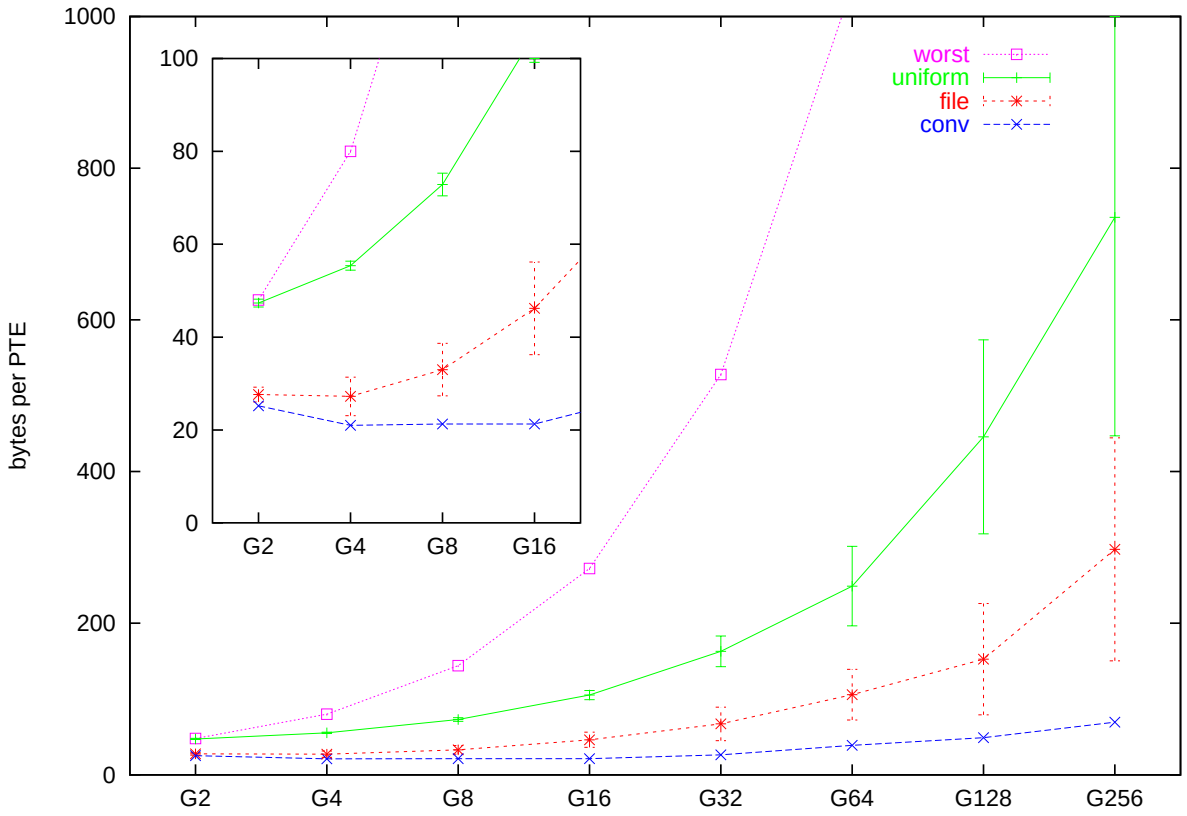


Address space establishment/teardown cost



Other benchmarks

- sparse benchmark: uniformly distributed pages in 1 T address space
- file benchmark: uniformly distributed multi-page objects



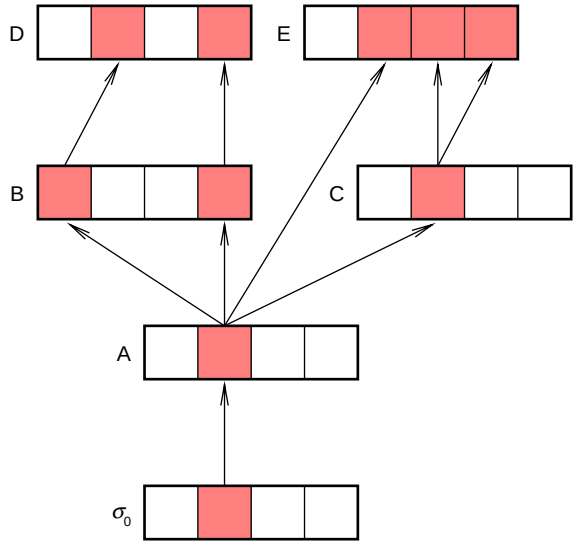
GPT conclusions

- low establishment/teardown cost
- small GPT node size saves space, especially for sparse distributions
- tree depth can become a problem, especially for dense distributions

L4/MIPS solution: use GPT with a level 2 TLB.

Implementation in L4

L4 provides three operations: **map**, **grant**, and **unmap**.



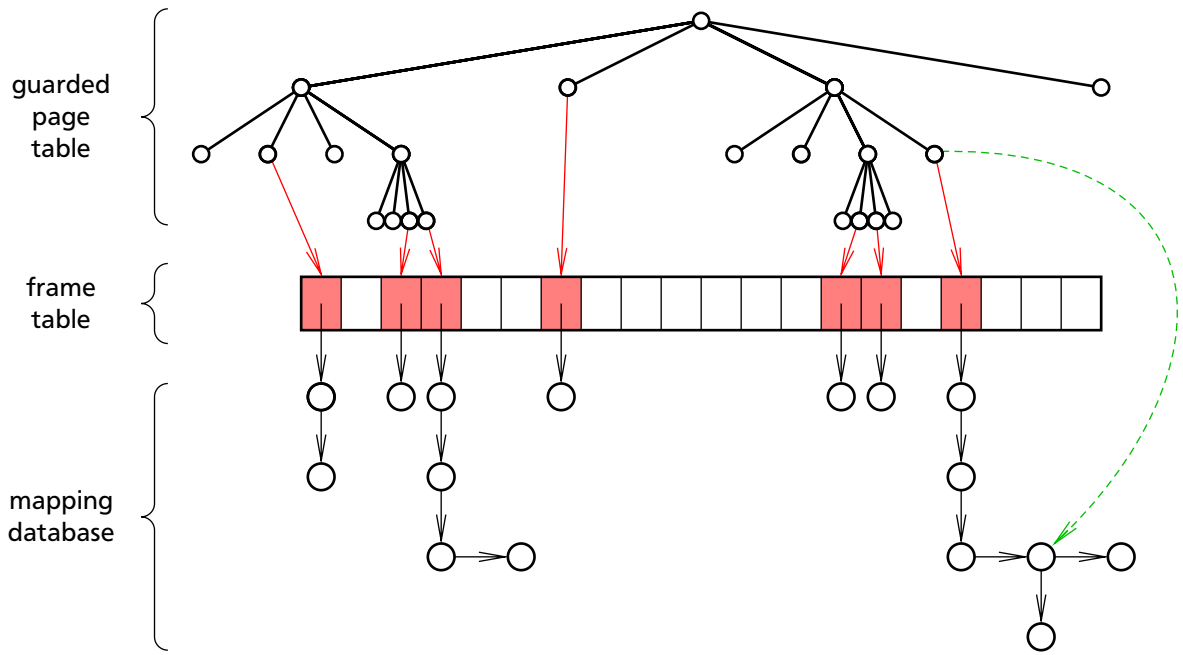
L4 must remember the history of **map** operations in the **mapping database**, to allow future undo with **unmap**.

Memory management and I/O is the responsibility of *user-level pagers*.

L4 implementation

Most L4 implementations (including L4/MIPS) have a similar implementation of recursive address spaces.

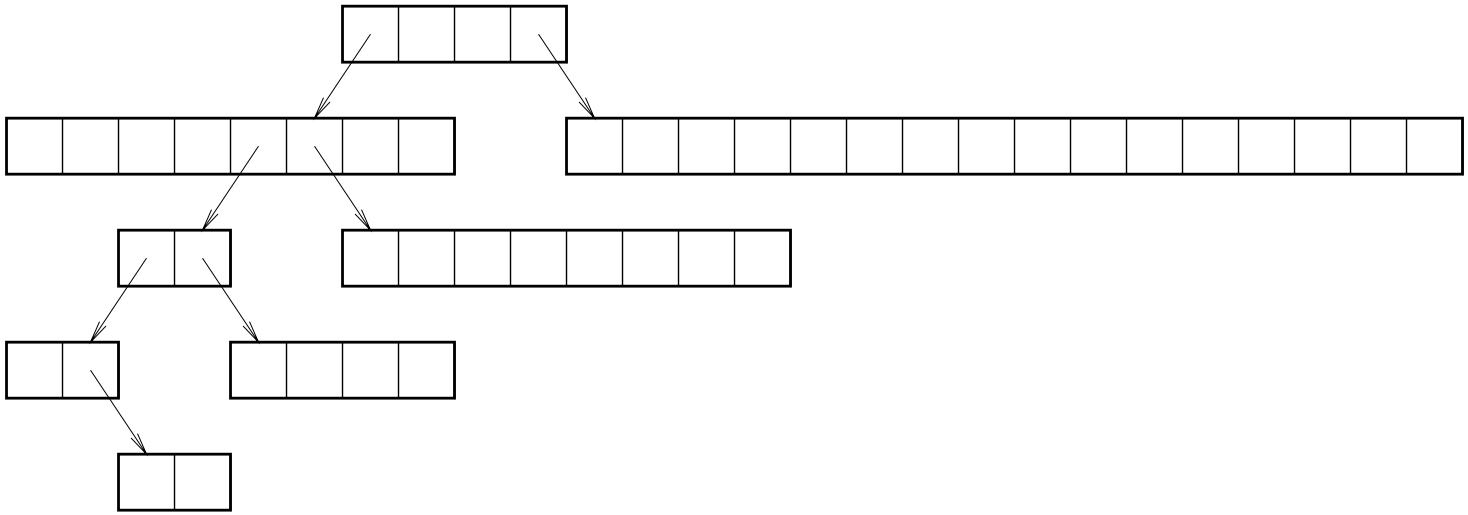
- guarded page table
- frame table
- mapping database



Direct pointers between GPT and mapping database (green arrow) were considered by Elphinstone, but rejected to allow PT implementation freedom.

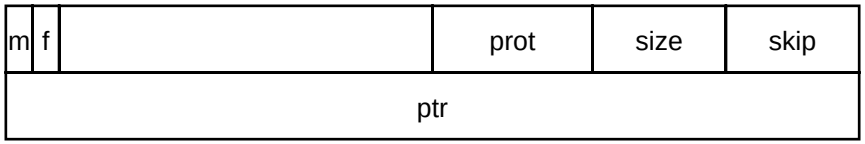
Level and Path Compressed Trie

- invented by Andersson and Nilsson (1991)
- implemented by Szmajda in Calypso VM system
- a simplified and flattened version of GPT
- allows node size and skip size to be an arbitrary power of two
- all guard comparison deferred until the leaves

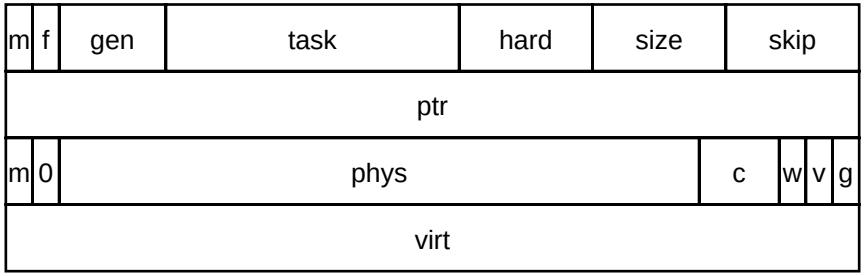


Calypso implementation

- store two shift amounts and a pointer in internal nodes
- extract bits with two shifts



- store virtual address (and other goodies) in an enlarged PTE



Each PTE may represent any (hard) page size.

Page tables may be shared (with an addition to the L4 API).

Calypso vs. GPT

Elphinstone (1999) derived the following GPT algorithm.

repeat {

$u = v \gg (v_{len} - s)$

$g = (p + 32u) \rightarrow guard$

$g_{len} = (p + 32u) \rightarrow guard_{len}$

if $g == (v \gg (v_{len} - s - g_{len}))$ **and** $(2^{g_{len}} - 1)$ {

$v'_{len} = v_{len} - s - g_{len}$

$v' = v$ **and** $(2^{g_{len}})$

$s' = (p + 32u) \rightarrow size'$

$p' = (p + 32u) \rightarrow table'$

} **else**

page fault

} **until** p is a leaf

Calypso vs. GPT

After common subexpression elimination, the GPT loop has 17 arithmetic and load operations.

Calypso is much simpler.

repeat {

$p = \&p \rightarrow ptr[v \ll p \rightarrow skip \gg p \rightarrow size]$

} **until** p is a leaf

if $p \rightarrow virt \neq v$

page fault

All guard checking is deferred until the end.

The inner loop on the MIPS R4000 requires only 7 instructions.

Calypso policies

How large to make Calypso nodes?

Andersson and Nilsson (1998) used thresholds like 50% full $\rightarrow$ double, etc. Calypso's implementation is different.

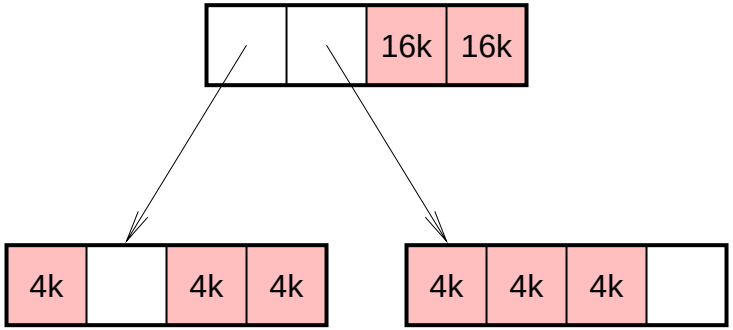
- each page table is greedy, and takes all the memory it can
- unused page tables are liable to be chopped in half at any time, and the returned to the memory manager
- power of two regions are managed by a buddy system allocator

To prevent excess greed, kernel memory is managed by user-level pagers, instead of in a single fixed pool.

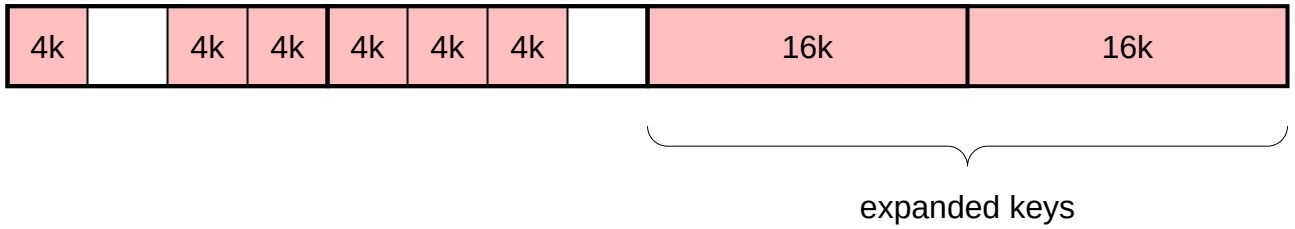
Status: memory management API undergoing standardization.

Page table fragmentation

But, representing many page sizes can blow out depth.



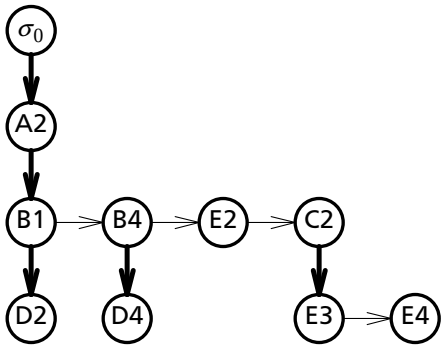
Solution: **key expansion**

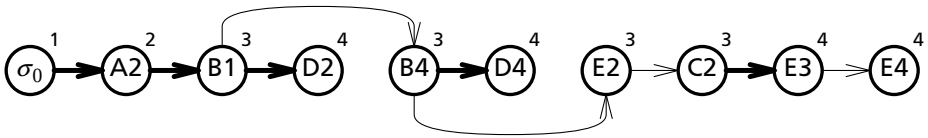


Complicates the mapping database (later).

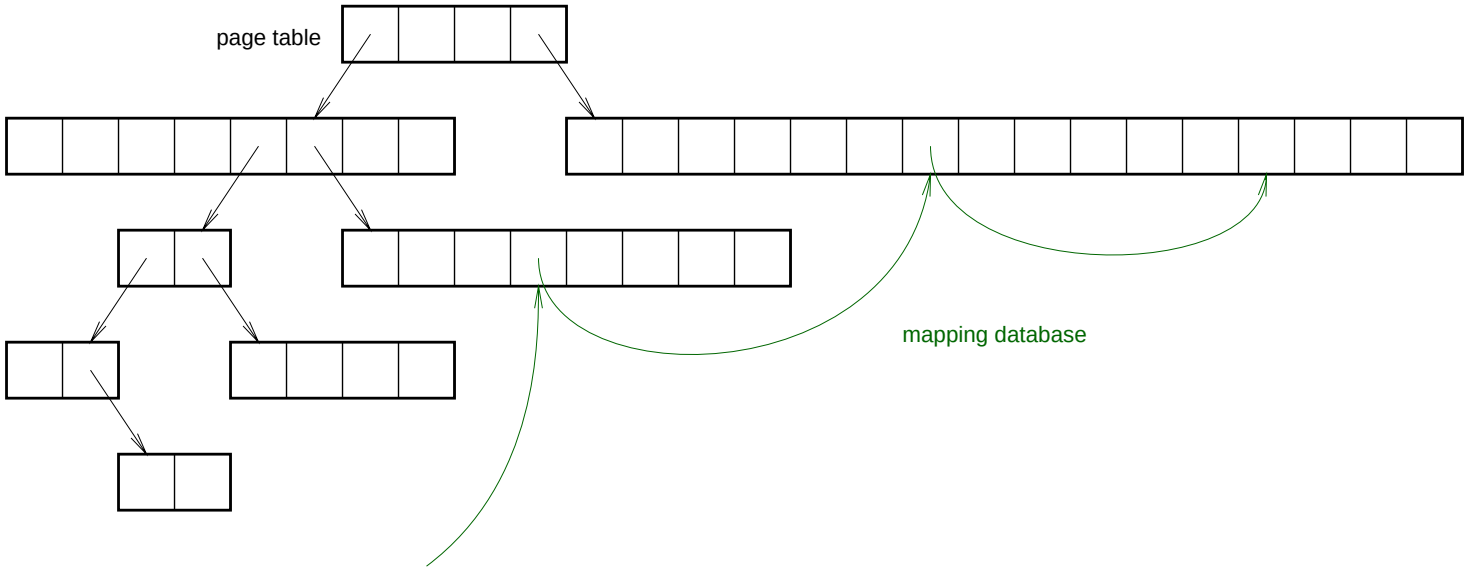
Calypso mapping database

Topologically sort each mapping graph into a singly-linked list.





Integrate the mapping database list into the PTEs.



Link

New VM operation: **link**, which establishes a shared domain between pager and pagee.

Semantically, **link** is like **map**, but instead of just copying a snapshot of the pager's mappings to the destination address space, the pager and pagee always share the mappings, even if the pager's address space is updated by future **map**s or **unmap**s.

L4 primitive	Unix analogy
unmap	`rm`
map	`cp`
grant	`mv`
link	`ln -s`

More on link

Restrictions:

- virtual address of the fpage in pager and pagee must be equal
- fpage size may be restricted

Advantages:

- natural generalization of **map** and **grant**
- reduces kernel crossings
- reduces page fault IPC
- restricted by L4's usual IPC confinement model (e.g. clans and chiefs)

Calypso performance

Results measured by running with VM on and off, and comparing run-times.

- counts all direct and indirect costs of VM
- normalized to percentage overhead

Calypso also includes other optimizations beyond the scope of this lecture.

	HPT	CPT	GPT	GPT+TLB2	CALY4
wave5	15.4%	14.9%	16.2%	5.1%	6.2%
swim	4.7%	2.4%	1.1%	0.5%	2.6%
gcc	24.3%	26.8%	31.4%	9.1%	9.5%
compress	16.2%	17.2%	24.5%	7.9%	7.6%

Single-tasking performance

Calypso performance

Enabling page size mixtures drastically improves performance; but space/time tradeoff is harder to measure.

	CALY4	CALY64	CALY1024	CALY16384
wave5	6.2%	2.4%	<0.1%	<0.1%
swim	2.6%	1.1%	0.0%	0.0%
gcc	9.5%	0.8%	0.0%	0.0%
compress	7.6%	2.6%	<0.1%	<0.1%

Mixed page sizes (assuming infinite physical memory)

Multi-tasking performance was measured with and without LINK, and using the G (global) bit to simulate shared TLB tags.

	GPT	GPT+TLB2	CALY4M	CALY4L	CALY4G
wave5	20.2%	9.1%	8.2%	8.0%	7.6%
swim	2.6%	2.1%	2.9%	2.8%	2.8%
gcc	36.9%	13.4%	11.8%	11.5%	10.8%
compress	27.9%	10.1%	9.1%	8.6%	8.3%

Multi-tasking performance (assuming infinite physical memory)

Conclusions

- Modern hardware and recent software can lead to high VM overhead.
  - 64-bit addresses
  - sparse address space usage
  - micro-kernel service decomposition
  - bloated applications
- Conventional page tables don’t perform well in these conditions.
- Level 2 TLB is the best solution to a slow page table
- Calypso performs as well as level 2 TLB for dense address spaces
- Performance in sparse situations yet to be evaluated
- Optimization of the critical path pays off
  - but **only** after evaluation and measurement.

References and further information

http://www.cse.unsw.edu.au/~cls/